Rozwiązania wzorcowe zadań (18 III 2020)

Zadania implementacyjne

Zadanie 1. Proszę zaimplementować algorytm QuickSort (do sortowania tablicy liczb) w sposób iteracyjny (czyli bez korzystania z rekurencji). Należy w tym celu wykorzystać samodzielnie skonstruowany stos. Należy założyć, że dostępne są nastpujące operacje:

S = Stack() # tworzy obiekt stosu
S.push(x) # umieszcza x na szczycie stosu
S.pop() # zdejmuje element ze szczytu stosu i zwraca go (jeśli stos jest pusty, to zgłaszany jest błąd---nie należy do tego dopuścić)
S.is_empty() # zwraca True jeśli stos jest pusty, False w przeciwnym wypadku

partition(A, i, j) # wykonuje operację partition z QuickSorta na obszarze tablicy od A[i] do A[j] (włącznie) i zwraca indeks k taki, że elementy A[i], ..., A[k-1] mają wartości mniejsze lub równe A[k], a elementy A[k+1], ..., A[j] (gdzie n to długość tablicy A) mają wartości większe od A[k].

Rozwiązanie wzorcowe (autor: Mateusz Praski)

```
def QuickSort( A, i, j ):
# posortuje fragment tablicy A[i], ..., A[j]
    S = Stack()
    S.push((i,j))
    while not S.is_empty():
        sub = S.pop()
        q = partition(A, sub[0], sub[1])
        if q-1 > sub[0]:
            S.push((sub[0], q-1))
        if q+1 < sub[1]:
            S.push((q+1, sub[1]))
```

Zadanie 2. Proszę zaimplementować algorytm QuickSort tak, żeby zużywał najwyżej O(log n) pamięci (poza pamięcią na tablicę do posortowania), niezależnie od tego jakie podziały wskaże funkcja partition. Proszę zwrócić uwagę, że jeśli funkcja partition będzie zawsze dzielić tablicę wejściową (o rozmiarze n) na części o rozmiarze n-1 (mniejsze lub równe pivotowi) i 1 (pivot) to standardowe implementacja rekurencyjna zużyje O(n) pamięci na stosie. Należy tak pokierować rekurencją, żeby zużyć najwyżej O(log n) pamięci.

Funkcja partition dostępna jak wyżej.

Rozwiązanie wzorcowe (autorka: Anna Gut)

```
def QuickSort( A, i, j ):
   while( i < j ):
      k = partition( A, i, j )
      if k-i < j-k:
         QuickSort( A, i, k-1 )
         i = k+1
      else:
         QuickSort( A, k+1, j )
         j = k-1
```

Zadanie 3. Proszę zaimplementować funkcję `Select( A, k )`, która zwraca k-ty co do wielkości element z tablicy A. Funkcja powinna działać w oczekiwanym czasie liniowym (przy założeniu, że funkcja partition z algorytmu QuickSort zawsze dzieli tablicę wejściową na dwie z grubsza równe części).

Dostępna jest funkcja partition jak wyżej.

Rozwiązanie wzorcowe (autor: Kamil Delekta)

```
def Select(A, k):
  return _Select(A, 0, len(A) - 1, k)

def _Select(arr, left, right, k):
  if left == right:
    return arr[left]
  q = partition(arr, left, right)
  size = q - left + 1

  if size == k:
    return arr[q]
  if k < size:
    return _Select(arr, left, q - 1, k)
  else:
    return _Select(arr, q + 1, right, k - size)
```

Zadanie 4. Dana jest jednokierunkowa lista odsyłaczowa, realizowana z elementów klasy `Node`:

```
class Node:
  def __init__(self):
     self.next = Null
     self.val  = Null
```

Lista L jest dostępna w postaci listy [first,last] zawierającej wskazania na pierwszy i ostatni element listy. Proszę zaimplementować funkcję `QuickSort( L )` sortującą listę L i zwracającą posortowaną listę także w postaci listy [first,last].

```
Null = None

### Wersja z listami pythonowymi - łatwiej zrozumieć ideę
# proszę zwrócić uwagę jak brak komentarzy utrudnia zrozumienie
# czyjegoś kodu i pamiętać o tym przy pisaniu kolokwium

def QuickSort( l ):
  if len(l)==0: return l
  x = l[0]
  lt = []
  eq = []
  gt = []
  while len(l)>0:
      y = l.pop(0)
      if y<x: lt.append(y)
      elif y>x: gt.append(y)
      else: eq.append(y)
  end
  return QuickSort(lt)+eq+QuickSort(gt)

l = [2,7,3,17,13,19,2,5,11,7]
l2 = QuickSort(l)
print(l2)
```

```
### wersja z listami odsyłaczowymi

class Node:
  def __init__(self):
      self.next = Null
      self.val  = Null
  end
end


def QuickSortll( l ):
  if empty(l): return l
  x = l[0].val
  lt = [Null,Null]
  eq = [Null,Null]
  gt = [Null,Null]
  while not empty(l):
      y = get(l)
      if y.val<x: app(lt,y)
      elif y.val>x: app(gt,y)
      else: app(eq,y)
  end
  return concat(concat(QuickSortll(lt),eq),QuickSortll(gt))
end

def concat(l1,l2):
  if l1[0]==Null: return l2
  if l2[0]==Null: return l1
  first = l1[0]
  l1[1].next = l2[0]
  last = l2[1]
  return [first,last]
end

def get(l):
  res = l[0]
  l[0] = l[0].next
  res.next = Null
  return res
end

def empty(l):
  return l[0]==Null
end

def app(l,el):
  if l[0]==Null: l[0] = el
  else: l[1].next = el
  l[1] = el
end

def printl(l):
  p = l[0]
  while p!=Null:
      print(p.val,end=' ')
      p = p.next
  end
  print()
end
```

```
def create(l):
  if len(l)==0: return [Null,Null]

  first = Node()
  first.val = l[0]
  first.next = Null
  last = first

  for i in range(1,len(l)):
      p = Node()
      p.val = l[i]
      p.next = Null
      last.next = p
      last = p
  end
  return [first,last]
end


l1 = [2,7,3,17,13,19,2,5,11,7]

ll1 = create(l1)
printl(ll1)
ll2 = QuickSortll(ll1)
printl(ll2)
```

Zadania opisowe

Zadanie 1. Dana jest tablica A o rozmiarze n, zawierająca liczby ze zbioru {0, ..., n^2-1}. Proszę podać algorytm sortujący tę tablicę w czasie O(n).

Rozwiązanie: Stosujemy sortowanie pozycyjne (radix sort), ale podstawą naszego systemu liczenia nie jest 10, tylko n. W związku z tym musimy wykonać dwa sortowania z użyciem sortowania przez zliczanie (counting sort), które wykona się w czasie O(n).

Zadanie 2. Proszę zaproponować algorytm, który na wejście dostaje tablicę A słów (być może o różnych długościach) i sortuje je w porządku słownikowym w czasio O(L), gdzie L to suma długości słów.

Rozwiązanie: Najpierw sortujemy wyrazy po długości (przez zliczanie). Następnie sortujemy wyrazy pozycyjnie z tą poprawką, że dla każdej pozycji 'i' znamy 'k' takie, że tab[:k] zawiera wyrazy krótsze od 'i'. I na pozycji 'i' sortujemy jedynie fragment tab[k:]. Gdy wyrazy są posortowane po długościach, to indeksy 'i' możemy wyznaczyć w czasie O(długość najdłuższego słowa). Całość działa, bo w sortowaniu pozycyjnym używamy sortowania stabilnego.

Zadanie 3. Proszę zaproponować algorytm, który mając na wejściu dwa słowa, x i y, składające się z małych liter alfabetu łacińskiego sprawdzi, czy słowa te są anagramami (czyli zawierają tyle samo, tych samych liter).

Rozwiązanie: Najpierw sprawdzamy czy słowa są jednakowej długości (jak nie, to słowa nie są anagramami). Tworzymy tablicę liczników L o rozmiarze alfabetu (nie musimy całej tablicy zerować)
Przebiegamy po słowie x zerując liczniki dla liter w słowie x - O(N)
Przebiegamy po słowie y zerując liczniki dla liter w słowie y - O(N)
Przebiegamy po słowie x zwiększając o jeden liczniki dla liter w słowie x - O(N)
Przebiegamy po słowie y zmniejszając o jeden liczniki dla liter w słowie y - O(N)
Przebiegamy po słowie x sprawdzając czy liczniki dla liter w słowie x są równe zero - O(N)
Przebiegamy po słowie y sprawdzając czy liczniki dla liter w słowie y są równe zero - O(N)
Jeżeli wszystkie liczniki dla liter x i y są równe zero słowa są anagramami

Zadanie 4. Pewien eksperyment fizyczny generuje bardzo szybko stosunkowo krótkie ciągi liczb całkowitych z przedziału od 0 do 10^9-1. Pomiar w eksperymencie polega na okresleniu ile różnych liczb znajduje się w danym ciągu. Niestety liczby są generowane tak szybko, że konieczne jest zagwarantowanie czasu działania rzęduO(1) na każdy element ciągu (pamięć jest dużo mniej krytycznym zasobem). Ciągi są generowane błyskawicznie, jeden po drugim. Proszę zaproponować strukturę danych pozwalającą na wykonywanie następujących operacji:

- init() - przygotowuje strukturę danych do pracy
- insert(x) - wstawia x do struktury; zwrace True, jeśli x nie była jeszcze wstawiona i False w przeciwnym razie
- reset() - usuwa wszystkie liczby

Wszystkie operacje powinny działać w czasie O(1) (pomijając koszt alokacji pamięci w funkcji init()).

Rozwiązanie: Tworzymy dwie tablice:

- A – tablica (indeksowana od 0 do 10^9-1). Każde pole A[i] ma dwa pola: A[i].reported (informacja czy liczba i się pojawiła) oraz A[i].ptr (wskaźnik)
- S – tablica o rozmiarze równym maksymalnej długości ciągu w eksperymencie, używana jako stos na wskaźniki

init(): alokuje te dwie tablice i zapisuje, że stos S jest pusty. Wymaga to czasu O(1) (nie licząc systemowego czasu na alokację pamięci)

reset(): zapisuje, że stos S jest pusty (czas O(1))

insert(x): Idea jest taka, że początkowo A[x].reported powinno mieć wartość False. Widząc to, wpisalibyśmy A[x].reported = True i zwrócili False. Gdyby A[x].reported = True, to zwrócilibyśmy False. Ale to nie do końca działą, bo nie możemy wyzerować tablicy A. Dlatego stosujemy strategię, w której możemy zweryfikować, czy A[x].reported ma poprawną wartość.

1. Sprawdzamy, czy A[x].ptr wskazuje na obszar tablicy S, w której przechowywane są elementy stosu. Jeśli nie, to wpisujemy do A[x].reported wartość False (bo nie widzieliśmy jeszcze wcześniej liczby x). Następnie umieszczamy na stosie S wskaźnik na A[x], a A[x].ptr wskaźnik do tego właśnie dodanego elementu stosu.
2. Jeśli A[x].ptr wskazuje na obszar stosu, to sprawdzamy czy znajdujący się tam wskaźnik z powrotem wskazuje na A[x]. Jeśli tak, to A[x].reported ma poprawną wartość. Jak nie, to wpisujemy do A[x].reported wartość False i postępujemy jak wyżej (dodajemy odpowiedni wskaźnik na stos i w A[x[.ptr umieszczamy wsaźnik zwrotny)
3. Po wykonaniu powyższych, wiemy że A[x].reported ma poprawną wartość.

Rozwiązanie opiera się na tym, że w obszarze stosu zawsze mamy poprawne wartości. Tak więc element A[x] musi „udowodnić", że ma poprawną wartość przez wskazanie „o ten element na stosie mówi, że jestem poprawny". Jeśli element na stosie faktycznie to mówi (jest tam wskaźnik zwrotny) to element A[x] poprawny jest. A jak nie, to nie.

Zadanie 5. Proszę podać możliwie jak najszybszy algorytm sortujący tablicę A liczb wymiernych. Tablica A zawiera tylko log(n) różnych wartości (gdzie n to długość tablicy A)

Rozwiązanie: Tu możliwe są różne warianty. Najbardziej oczywisty i zaproponowany przez większość uczestników polega na tym, że konstruujemy posortowaną tablicę krotek (powiedzmy B), gdzie pierwszy element to wartość, a drugi to liczba jej wystąpień w tablicy A. Skanujemy A i dla każdego elementu szukamy jego odpowiednika w B, stosując wyszukiwanie binarne. Jak nie znajdziemy, qw B danej liczby to dodajemy ją do B, a jak znajdziemy – zwiększamy licznik liczby wystąpień. W pierwszym przypadku wyszukiwanie ma złożoność O(log(length(B))), czyli O(log log n). W drugim – szereg elementów trzeba przesunąć, ale skoro takich przesunięć nie może być więcej niż logn, to ogólna złożoność tej części nadal wynosi O (n log log n) (łączny koszt przesunięć to $O(\log^2 n)$).

Podobny efekt można uzyskać za pomocą innych struktur danych, takich jak np. drzewa czerwono-czarne albo AVL (generalnie chodzi o to żeby można było w takich strukturach wyszukać element w czasie logarytmicznym). Ale na wykładach takich struktur jeszcze nie było.

Inne możliwe podejście (nikt go nie zgłosił) mogłoby polegać na zastosowaniu algorytmu QuickerSort (wersji QuickSort gdzie dzielimy tablicę na mniejsze od pivota, równe pivotowi i większe od pivota). Algorytm zagłębi się rekurencyjnie tylko na loglogn poziomów, co daje złożoność O(nloglogn) (zakładając, że będziemy mieli dobre wybory pivotów).

Zadanie 6. Dana jest tablica A liczb wymiernych (wszystkie liczby są różne). Proszę podać algorytm, który znajdzie takie dwie liczby A[i] i A[j], które po posortowaniu tablicy występują bezpośrednio koło siebie i których różnica jest maksymalna.

Rozwiązanie: Zakładamy, że A ma n liczb.

1. Najpierw znajdujemy minimum i maksimum w A (w czasie O(n))
2. Tworzymy n kubełków i rozrzucamy liczby z A do kubełków (jak w sortowaniu kubełkowym; złożoność O(n))
3. Jeśli każdy kubełek zawiera co najmniej jedną liczbę (a konkretnie---dokładandie jedną liczbę) to mamy dane posortowane i po prostu znajdujemy wynik w jednym liniowym przeglądzie (czas O(n))
4. Jeśli któryś kubełek jest pusty, to znaczy że A[i] jest minimum z któregoś kubełka a A[j] to maksimum z któregoś kubełka. Obliczam minimum i maksimum dla wszystkich kubełków i znajduję wynik przeglądając po kolei kubełki.

Zadanie 7. Problem pojemników z wodą—zadanie z pewnego konkursu programistycznego.

Mamy serię pojemników z wodą, połączonych (każdy z każdym) rurami. Pojemniki maja kształty prostokątów (2d), rury nie maja objętości (powierzchni). Każdy pojemnik opisany jest przez współrzędne lewego górnego rogu i prawego dolnego rogu. Wiemy, że do pojemników nalano wody (oczywiście woda rurami spłynęła do najniższych pojemników). Proszę podać algorytm Obliczający ile pojemników zostało w pełni zalanych.

Rozwiązanie (podane przez Pana Kamila Kurowskiego):
Stwórzmy strukture która umożliwi nam efektywne działanie na pojemnikach.
Kazdy pojemnik bedziemy reprezentować jako 2 punkty, a kazdy punkt jako liste 3 elementow [y, szerekosc, flaga], gdzie y to wspolrzedna y, szerekosc to szerekosc pojemnika, a flaga to informacja czy to jego dolny, czy gorny punkt.
Strukture nastepnie sortujemy wzgledem y, i przezchodzimy cala tablice wykonujac nastepujace operacje:
pamietamy ogolna szerekosc pojemnikow,
- jesli punkt jest poczatkiem pojemnika, obliczamy ilosc wody, i zwiekszamy szerokosc
- jesli punkt jest koncem pojemnika, obliczamy ilosc wody, i zmiejszamy szerokosc
ilosc wody to szerekosc * wysokosc, gdzie wysokosc to roznica miedzy y i-tego ora (i - 1) elementu

za kazdym razem ilosc wody odejmujemy od calkowitej wartosci, a gdy jestemy w punkcie konca i calkowita ilosc wody jest nieujemna zwiekszamy licznik pojemnikow

zlozonosc O(nlogn) lub O(n) w zaleznosci od sortowania

Dopowiedzenie: brakuje rozwiązania kwestii, gdzie może istnieć kilka takich samych pojemników (np gdyby to była siatka wyglądająca jak Manhattan)

Istnieje także rozwiązanie, które nie wykorzystuje sortowania (ale używa wyszukiwania binarnego)